

UNITED STATES PATENT APPLICATION
FOR

**OPERAND CONVERSION
OPTIMIZATION**

INVENTOR:

RICHARD FORD,
a citizen of the United States

PREPARED BY:

BLAKELY, SOKOLOFF, TAYLOR & ZAFMAN LLP
12400 WILSHIRE BOULEVARD
SEVENTH FLOOR
LOS ANGELES, CA 90025-1026
(303) 740-1980

EXPRESS MAIL CERTIFICATE OF MAILING

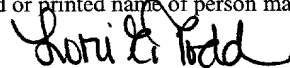
"Express Mail" mailing label number: EL 866506867 US

Date of Deposit: September 28, 2001

I hereby certify that I am causing this paper or fee to be deposited with the United States Postal Service "Express Mail Post Office to Addressee" service on the date indicated above and that this paper or fee has been addressed to the Commissioner of Patents and Trademarks, Washington, D.C. 20231.

Lori E. Todd

(Typed or printed name of person mailing paper or fee)



(Signature of person mailing paper or fee)

September 28, 2001

(Date signed)

OPERAND CONVERSION OPTIMIZATION

COPYRIGHT NOTICE

[0001] Contained herein is material that is subject to copyright protection. The copyright owner has no objection to the facsimile reproduction by anyone of the patent document or the patent disclosure, as it appears in the United States Patent and Trademark Office patent file or records, but otherwise reserves all rights to the copyright whatsoever. The following notice applies to the software and data as described below and in the drawings hereto: Copyright © 2001, Intel Corporation, All Rights Reserved.

FIELD OF THE INVENTION

[0002] This invention relates to computers in general, and more specifically to optimization of applications by the elimination of redundant operand conversions.

BACKGROUND OF THE INVENTION

[0003] In computer operations, operands may require conversion between different formats. For example, an operand may be stored in one format but certain operations may be conducted in another format. A specific example of operand conversions regards floating point calculations. In certain computing devices, because of considerations such as size, power, and expected usage, a floating point logic unit for the execution of floating point mathematical functions is not included. If floating point operations are needed, the calculations may be done by software emulation of floating point operations.

[0004] However, conversions of operands between formats will generally slow system operations. Software emulation of floating point operations are executed

relatively slowly. The device for which software emulation is applied, because it lacks floating point hardware, cannot effectively conduct mathematical operations on floating point operands while the operands are in floating point form. For this reason, a conventional software emulation of a floating point operation results in converting each operand into another format that is more easily operated upon. A standard floating point number is stored in a relatively small format, and is said to be in a “packed” format. A number in a “packed” floating point format may be converted to an “unpacked” format that can be effectively used in calculations.

[0005] In conventional operand conversions, the operand is converted from a first format to a second format for the purposes of an operation and then is converted back to the original format after the operation. In floating point emulation, operands are generally stored in packed format, are converted into unpacked format for the execution of floating point calculations, and then the result of the function is packed again after the calculation.

[0006] The process of packing and unpacking numbers for floating point operations, while allowing for more efficient mathematical operations in a computing environment in which a floating point logic unit is absent, also can take significant time and thus impose performance costs. As calculations become more complex, there may be results of operations that are packed after the operations performed, and then that are subsequently unpacked again if the results are operands for additional operations. The format conversions may create unnecessary delays because of the redundant transformations of operands between the packed and unpacked formats.

BRIEF DESCRIPTION OF THE DRAWINGS

[0007] The appended claims set forth the features of the invention with particularity. The invention, together with its advantages, may be best understood from the following detailed descriptions taken in conjunction with the accompanying
5 drawings, of which:

[0008] Figure 1A illustrates the structure of floating point numbers under IEEE Standard 754;

[0009] Figure 1B illustrates the storage of a single precision floating point number under IEEE Standard 754;

10 [0010] Figure 2 illustrates an unpacked floating point number under a particular embodiment;

[0011] Figure 3A is a conventional addition routine for floating point emulation;

[0012] Figure 3B is a conventional subtraction routine for floating point emulation;

15 [0013] Figure 4 illustrates the expansion and computation of an exemplary floating point equation under conventional floating point emulation;

[0014] Figure 5 is a block diagram illustrating an embodiment of operand conversion optimization;

[0015] Figure 6 is a block diagram of an embodiment of a exemplary
20 compilation system including operand conversion optimization;

[0016] Figure 7 is a block diagram illustrating an embodiment of optimization of floating point emulation;

[0017] Figure 8 illustrates the expansion, optimization, and computation of a exemplary floating point equation;

[0018] Figure 9A is a diagram of an exemplary loop containing a partial redundancy; and

5 [0019] Figure 9B is a diagram of an optimized loop in which the partial redundancy has been eliminated.

0995701.092801

DETAILED DESCRIPTION

[0020] A method and apparatus are described for optimization by the elimination of redundant operand conversions. In an embodiment, a series of functions is examined to determine whether redundant sequences converting operands between a plurality of formats exist. The redundant sequences are eliminated to optimize the operation of the functions.

[0021] In the following description, for the purposes of explanation, numerous specific details are set forth in order to provide a thorough understanding of the present invention. However, it will be apparent to one skilled in the art that the present invention may be practiced without some of these specific details. In other instances, well-known structures and devices are shown in block diagram form.

[0022] The present invention includes various processes, which will be described below. The processes of the present invention may be performed by hardware components or may be embodied in machine-executable instructions, which may be used to cause a general-purpose or special-purpose processor or logic circuits programmed with the instructions to perform the processes. Alternatively, the processes may be performed by a combination of hardware and software.

Terminology

[0023] Before describing an exemplary environment in which various embodiments of the present invention may be implemented, some terms that will be used throughout this application will briefly be defined:

[0024] As used herein, “packed” indicates that a number is stored in a floating point format, including but not limited to the single precision and double precision

floating point formats pursuant to Standard 754 of the Institute of Electrical and Electronic Engineers, *IEEE Standard for Binary Floating-Point Arithmetic*, IEEE Computer Society, published August 12, 1985.

[0025] "Unpacked" indicates that a floating point number is stored in converted format. A floating point number may be converted to an unpacked format to enable floating point mathematical operations in a computing environment in which a floating point logic unit is not present. While the examples provided herein provide for an "expanded" format in which a floating point number requires greater storage than in packed format, embodiments are possible in which the size of the storage is not increased.

[0026] An operand is a "floating point" number when the operand is expressed as a mantissa multiplied by a radix, where the radix is raised to the power of an exponent. A general form of a floating point number expressed in mathematical terms is:

$$m \times r^e \quad [1]$$

where m = mantissa, r = radix, e = exponent

[0027] Under an embodiment, an application includes a series of operations. In order to perform a first operation, one or more operands of the first operation are converted from a first format to a second format. After the first operation is performed, the result is converted back from the second format to the first format. However, the result of the first operation may be an operand for a second operation. In conventional functions, the result of the first operation is again converted from the first format to the second format in order to perform the second operation. The conversion of the result from the second format to the first format after the performance of the first operation

followed by conversion from the first format to the second format before the performance of the second operation thus creates a redundant sequence of conversions.

[0028] Under an embodiment, the operations included within an application are examined and the origin of each operand converted from a first format to a second is determined. If the origin of an operand is a conversion from the second format to the first format, then the redundant series of conversions is eliminated to optimize the performance of the application. Note that while the embodiment described relates to conversions between a first format and a second format, other embodiments might involve conversions between more than two formats, including conversions to and from intermediate formats. After redundant conversions have been eliminated, code may be partially compressed to save required code storage space sequence, which may be possible when certain operations do not have redundant conversion sequences to eliminate.

[0029] In a particular embodiment, an application includes software emulation of floating point operations. In a computing environment in which there is no floating point logic unit, it is generally more efficient to perform mathematical functions using fixed point numbers. In a software emulation of a floating point operation, floating point numbers are generally converted to fixed point format prior to performing the operation, with the result of the operation then converted from fixed point format back to floating point format. If the result of a first floating point operation is an operand of a second floating point operation, the number is then converted back to fixed point format prior to the performance of the second operation.

[0030] According to an embodiment, an application including floating point operations is expanded. The expansion of the application includes the conversion of floating point operands from a packed format into an unpacked format prior to performing floating point operations, and then the conversion of operation results from the unpacked format to the packed format subsequent to the performance of the operations. The operations of the application are examined to determine the origin of any operand that is converted from packed to unpacked format. If the origin of such an operand is a previous conversion of the number from the unpacked format to the packed format, the redundant pack-unpack conversion sequence can be eliminated to optimize the operation of the application.

[0031] In one embodiment, floating point numbers are expressed in packed form as IEEE Standard 754 floating point numbers. While embodiments are not limited to a particular standard for floating point representation, IEEE Standard 754 format is most commonly used for floating point format and is used herein for the purposes of illustration. IEEE floating point numbers are comprised of a sign, an exponent, and a mantissa and may be expressed either as single precision (32 bit) or double precision (64 bit) numbers. In general cases, a floating point number according to IEEE Standard 754 may be expressed as:

$$-1^S \times (1.m) \times 2^{(e-b)} \quad [2]$$

where: S is a sign bit equal to 0 for positive and 1 for negative;
 e is an exponent value with 8 bits for single precision and 11 bits for double precision;
 b is the bias, a number subtracted from the stored exponent to obtain the actual exponent, equal to 127 for single precision and 1023 for double precision;
 and
 m is the portion of the mantissa after the decimal point, with 23 bits for single precision and 52 bits for double precision.

1086907029660

[0032] **Figure 1A** illustrates the structure of single and double precision floating point numbers according to IEEE Standard 754. Floating point numbers are distinguished by the precision **105** of a number, with single precision numbers being stored in 32 bits and double precision numbers being stored in 64 bits. The sign **110** of a number is stored in a single bit, either bit 31 for single precision numbers or bit 63 for double precision numbers. The exponent **115** is stored in bits 30-23 (8 bits) for single precision numbers or bits 62-52 (11 bits) for double precision numbers. The mantissa **120** is stored in bits 22-0 (23 bits) for single precision numbers or bits 51-0 (52 bits) for double precision numbers. The bias **125**, as indicated above, is 127 for single precision numbers and 1023 for double precision numbers. **Figure 1B** provides a diagram of a single precision number under IEEE Standard 754, with 1 bit reserved for the sign **130**, 8 bits reserved for the exponent **135**, and 23 bits reserved for the mantissa **140**. There are special cases for the storage of certain numbers such as zero and infinity that are not described here.

15 [0033] **Figure 2** is an illustration of an unpacked floating point number under a particular embodiment. In this embodiment, the floating point number is expanded into a short integer (16 bits) for the sign **205**, a short integer for the exponent **210**, and a long integer (32 bits) for the mantissa **215**. A variant of this illustration is a mantissa that is signed, thereby eliminating the need for a variable containing the sign of the unpacked number. This is an example of a converted numerical format that may be used under a particular embodiment, but other embodiments may use other numerical formats.

20 [0034] A conventional floating point addition routine **305** is shown in **Figure 3A** and a conventional floating point subtraction routine **310** is shown in **Figure 3B**. The

0966701-0966701
routines shown in Figures 3A and 3B are part of a software floating point library created by the Free Software Foundation, Inc. of Boston, Massachusetts, copyright © 1994, 1995, 1996, 1997, and 1998. In the routines, floating point operands are unpacked prior to performing the operation, **315** and **320**. In the floating point addition routine **305** the
5 unpacked operands are added and a result designated as “res” is obtained **325**. In the floating point subtraction routine **310**, the sign of the second operand is first inverted **330** and then the unpacked operands are added and a result designated as res is obtained **335**. In both routines the unpacked result res is packed and returned as the result of the function, **340** and **345**. Other conventional software emulation routines, such as floating
10 point multiplication and floating point division, operate in a similar manner.

[0035] In Figure 4, a basic example of calculation utilizing conventional emulation of floating point mathematics is shown in pseudocode. Note that the example provided in Figure 4 and examples provided elsewhere herein are written assuming a static single assignment (SSA) naming format, in which there are distinct names for each
15 value of a variable. For example, if a calculation includes a definition of $X = 1$ followed by $X = X + 1$, the two “X” variables receive distinct names in the system. For the purposes of this explanation, the same equations may instead be referred to as $X_0 = 1$ followed by $X_1 = X_0 + 1$. Other name assignment formats may be utilized in other embodiments.

20 [0036] In Figure 4, the basic equation $T = a \times b \times c$ **405** multiplying three operands is resolved as a multiplication of two operands to obtain a first result followed by the multiplication of the first result times the third operand. This may be expressed as $T_0 = a \times b$ **410** and $T_1 = T_0 \times c$ **415**. These basic expressions are then expanded into the

commands required to evaluate the expressions. The first two operands are unpacked, $T_2 = \text{unpack}(a)$ **420** and $T_3 = \text{unpack}(b)$ **425**, and the unpacked operands are multiplied together, $T_4 = \text{unpack_mult}(T_2, T_3)$ **430**. The result is then packed into floating point format, $T_0 = \text{pack}(T_4)$ **435**. To complete the second half of the equation, for $T_1 = T_0 \times c$ **415**, the two operands are expanded, $T_5 = \text{unpack}(T_0)$ **440** and $T_6 = \text{unpack}(c)$ **445**. The expanded operands are then multiplied together, $T_7 = \text{unpack_mult}(T_5, T_6)$ **450**, and finally the product is packed into floating point format, $T_1 = \text{pack}(T_7)$ **455**. However, it can be seen that $T_0 = \text{pack}(T_4)$ **435** and $T_5 = \text{unpack}(T_0)$ **440** are unnecessary steps in that the result is packed after completion of the multiplication of T_5 and T_6 , followed by unpacking again to multiply the result times the third operand. The packing and unpacking of the number thus are redundant processes in conventional floating point multiplication.

[0037] Figure 5 is a block diagram illustrating a general embodiment of operand conversion optimization. In this embodiment, an application is received that includes the conversion of operands between formats, process block **505**. This may be a conversion of an operand from a first format to a second format before performing an operation and a conversion of the result of the operation from the second format to the first format after the performance of the operation. If necessary, the application is expanded into basic operations, process block **510**. An example of the expansion of an application into basic operations is the expansion of a complex equation into the basic mathematical functions required to calculate the equation, including any conversions of operands from the first format to the second format and any conversions of results from

the second format to the first format. Each operand that requires conversion from the first format to the second format prior to performance of an operation is examined and the origin of each such operand is determined, process block **515**. Any instances in which the origin of an operand is a conversion from the second format to the first format are identified, process block **520**. In these instances, there is a redundancy because the origin of the operand is the conversion of the number from the second format to the first format, which would occur if the operand was the result of a previous operation, and then the number is converted back to the second format before an operation is performed. The redundant operand conversions are then eliminated, process block **525**, which allows the application to run without performing the wasted steps. After optimization, partial compression of the operations, process block **530**, may reduce code storage size.

[0038] **Figure 6** illustrates an embodiment of the optimization of floating point conversions in the context of compiling a source file to obtain an object file. A source file, process block **605**, is initially presented to the front end of the compilation process, process block **610**, to parse the commands of the source code. The source file is converted into an intermediate representation, process block **615**, in which the code is represented with float operations in standard compressed representation without optimization. At this point, operand conversion redundancies may exist in the code. Following the conversion of the code into an intermediate representation is the conversion optimization sequence, **620**, comprising of expanding floating point values, process block **625**, in which floating point numbers are expressed in unpacked format; eliminating any pack/unpack redundancies that are discovered, process block **630**, to optimize the operation of the code; and partial compression of floating point values,

process block **635**, in which the resulting code is partially returned to compressed format to reduce code size. In the partial compression of floating point values, process block **635**, certain floating point operations have no conversion redundancies to optimize and thus can be compressed into the original compressed format. The resulting code may then subjected to further optimizing, process block **640**, and the optimized code is processed by a code generator, process block **645**, to generate the assembly language code for the application and an assembler, process block **650**, to generate the machine language code for the resulting object file, process block **655**.

[0039] In Figure 7, an embodiment of optimization of floating point emulation is illustrated. In the embodiment, an application including floating point calculations is received, process block **705**, and the calculations are expanded into floating point emulation operations, process block **710**. The operations are examined to determine the origin of the operands, process block **715**. Redundant pack-unpack sequences are identified when the origin of any operand is a pack function, process block **720**. As indicated above, a redundant pack-unpack sequence is a sequence in which a number is packed and then is subsequently unpacked for use in another operation. Redundancies are removed from the emulation operations, process block **725**, and operations are partially compressed, process block **725**, with compression possible when certain operations do not contain pack-unpack redundancies that can be eliminated.

[0040] Figure 8 is an illustration of an emulation of a floating point operation in which a redundant pack-unpack sequence has been eliminated. In this example, the basic equation $T = a \times b \times c$ **805**, as used in Figure 4, is again expanded. The equation may be expressed as $T_0 = a \times b$ **810** and $T_1 = T_0 \times c$ **815**. The multiplication of the first two

operands is performed by unpacking the floating point numbers, $T_2 = \text{unpack}(a)$ 820 and $T_3 = \text{unpack}(b)$ 825, and multiplying the unpacked operands, $T_4 = \text{unpack_mult}(T_2, T_3)$ 830. In order to multiply the product of T_2 and T_3 , designated as T_4 , with the third operand c , it is only necessary to unpack c , $T_5 = \text{unpack}(c)$ 835, since T_4 remains in unpacked format. The operation is thus completed by generating the product of the unpacked multiplication, $T_6 = \text{unpack_mult}(T_4, T_5)$ 840, and packing the result into floating point format, $T_1 = \text{pack}(T_6)$ 845. In this example, the removal of the redundant pack-unpack sequence eliminates two process steps from the expression shown in Figure 4.

[0041] A special case regarding operand conversion optimization occurs in situations of partial redundancy. In partial redundancy, there is a redundant conversion in a first case, but not in a second case. An example of partial redundancy occurs in a loop that uses one or more results from a loop cycle as operands for operations in a subsequent loop cycle. In this case, operand conversion may be redundant if the loop is repeated but may not be redundant if the loop is not repeated. It is known that invariants may be converted outside a loop instead of inside the loop to improve operations, but variables that change in value by operation of a loop are different in operation.

[0042] Figure 9A illustrates a loop containing a partial redundancy. In this particular example, floating point format conversions for software emulation of floating point operations are shown, but other types of variable conversions could exist in other embodiments. In this example, a loop contains the equation $a = a \cdot b$, which in static single assignment form may be expressed $a_2 = \Phi(a_1, a_3)$ followed by $a_3 = a_2 \cdot b$, where

a_1 is the value of a prior to the execution of the loop. In this pseudocode expression, the Φ -function chooses the appropriate value based on the control path used to reach the expression. In this case, a_1 is chosen in the initial loop and a_3 is chosen in subsequent loops. After some possible previous operations **905**, a_1 is set at a value **910**. A loop **915** contains the expanded form of the loop equation, including the required unpacking of the operands prior to multiplication and the subsequent packing after the multiplication operation. Following the loop **915**, there is a decision **920** to determine whether a statement is true. If the statement is not true, loop **915** is repeated. If the statement is true, the application path exits loop **915** and continues with other succeeding operations **925**. The operation $a_3 = \text{pack}(t_3)$ at the conclusion of the loop will form a redundant sequence with $t_1 = \text{unpack}(a_2)$ in a case in which the loop is repeated but not in a case in which the path exits the loop, thereby forming a partial redundancy.

[0043] Figure 9B illustrates a loop in which the partial redundancy shown in Figure 9A has been eliminated to optimize the operation. In this instance, following the previous operations and setting a_1 to a value, a_1 and b are unpacked **935**, with b being unpacked to move this invariant operation out of the loop and a_1 being unpacked to address the partial redundancy. Within the loop **940**, the operations are now performed in unpacked form with no packing or unpacking. If the following decision condition **945** is false, the loop continues without the redundancy. If the decision condition **945** is true, then a_3 is packed outside of the loop **950**, followed by any succeeding operations **955**. In this way, the packing operation is eliminated when it comprises part of a redundant pack-unpack sequence but is performed when there is no redundancy.

09966701.093604
[0044] The degree to which a floating point operation may be optimized by the elimination of redundant pack-unpack sequences is dependent on the actual functions and operands that are generated. The examples provided herein are purposely simple to express the basic concepts involved, but considerably more complex calculations may be optimized under various embodiments. Such examples may include a large number of floating point operations, floating point operations that are mixed with other types of operations, operations performed in multiple loops or as a part of iterative solutions, and other complicating factors.

10 [0045] In the foregoing specification, the invention has been described with reference to specific embodiments thereof. It will, however, be evident that various modifications and changes may be made thereto without departing from the broader spirit and scope of the invention. The specification and drawings are, accordingly, to be regarded in an illustrative rather than a restrictive sense.